SIEIS — Foolproof Implementation Plan

From Current State → Full ML-Powered Dashboard

Date: February 19, 2026 | **For:** Aishwarya (Group 8, AAI-530)

Where You Are Right Now (The Foundation)

Think of your project like building a house. You've already laid the **foundation and plumbing**:

```
✅ DONE (Your Foundation)
├── Docker Compose stack running (InfluxDB, MinIO, Kafka/Redpanda, Simulator, Consumer)
├── Historical data loader (1.3M+ records → InfluxDB + MinIO)
├── Real-time simulator (42 motes emitting sensor data through Kafka)
├── Consumer writing to both InfluxDB (hot) and MinIO (cold) storage
├── Verification scripts for both data stores
├── Project structure, .env config, requirements.txt, tests folder
└── Documentation (architecture, roadmap, deployment, ML docs)
```

```
🔲 TO BUILD (This Plan)
├── Phase A: Data Validation (the "health check")
├── Phase B: FastAPI Backend (the "brain")
├── Phase C: ML Model Training (the "intelligence")
├── Phase D: ML Inference API (the "prediction engine")
├── Phase E: Streamlit Dashboards (the "face")
├── Phase F: Incremental Retraining (the "learning loop")
└── Phase G: Integration & Polish (the "finishing touches")
```

Beginner Concepts You Need to Know

What is an ML Pipeline?

Analogy: A Factory Assembly Line

Imagine a chocolate factory:

1. **Raw cocoa beans arrive** → That's your raw sensor data in MinIO/InfluxDB
2. **Beans are cleaned, roasted, sorted** → That's *feature engineering* (preparing data for the model)
3. **A machine molds chocolates** → That's *model training* (the algorithm learns patterns)
4. **Quality inspector checks each chocolate** → That's *model evaluation* (checking accuracy)
5. **Chocolates are boxed and shipped** → That's *model deployment* (serving predictions via API)

6. **Customer feedback improves the recipe** → That's *incremental retraining* (model gets smarter over time)

What is Isolation Forest?

Analogy: The Odd-One-Out Game

Imagine you have 100 apples. 95 are red, 5 are blue. If you randomly pick features to split them, the blue ones get "isolated" very quickly (in just 1-2 splits), while red ones take many splits to separate. Isolation Forest works the same way — anomalies are data points that are easy to isolate. Short path in the tree = anomaly. Long path = normal.

Your sensor data has temperature, humidity, light, and voltage. If a mote suddenly reads 50°C when everything else is 22°C, Isolation Forest catches it because that reading gets isolated in very few splits.

What is Incremental Retraining?

Analogy: A Student Studying Weekly

Instead of cramming everything the night before an exam (training once on ALL data), the student studies a little each week (retraining on new batches). The model doesn't forget old knowledge — it gets updated with fresh patterns from recent data. In your case, you retrain manually whenever you load new sensor data.

Hot Path vs. Cold Path

Analogy: Your Kitchen vs. Your Pantry

- **Hot Path (InfluxDB):** Like your kitchen counter — recent items you use constantly. Fast access, limited space. Holds the last 30 days of sensor data for real-time monitoring.
- **Cold Path (MinIO):** Like your pantry — everything stored long-term in organized containers (Parquet files). Holds ALL historical data for training ML models and trend analysis.

Phase A: Data Validation (2–3 hours)

What: Make sure the data you loaded is clean and complete before building anything on top. **Why:** Building an ML model on bad data is like baking a cake with spoiled ingredients — the output will be garbage.

What to Build

One script: `scripts/validate_data_quality.py`

This script connects to BOTH InfluxDB and MinIO and checks:

Check	What It Means	Pass Criteria
Total record count	Did everything load?	1,500,000+ records
Duplicate ratio	Are there repeated readings?	Less than 1% duplicates
Mote coverage	Are all 42 sensors present?	42 unique mote IDs
Date range	Does the data span the right period?	Jan 19 – Feb 19, 2026
Value ranges	Are readings physically plausible?	Temp: -10 to 50°C, Humidity: 0-100%, Light: 0-500, Voltage: 2.0-3.0V
Missing values	How many gaps exist?	Less than 5% missing
InfluxDB vs MinIO match	Do both stores agree?	Record counts within 1% of each other

How to Build It in VS Code

1. Create the file at `scripts/validate_data_quality.py`
2. It should import `influxdb_client` and `minio` from your existing config
3. Run each check, collect pass/fail results
4. Print a summary report at the end

Expected Output

```
DATA QUALITY VALIDATION REPORT
═══════════════════════════════════════
✓ Total Records: 1,500,000
✓ Duplicates: 2,340 (0.16%) — PASS
✓ Unique Motes: 42/42 — PASS
✓ Date Range: 2026-01-19 to 2026-02-19 — PASS
✓ Missing Values: 45,000 (3%) — PASS
✓ InfluxDB ↔ MinIO Consistency — PASS
═══════════════════════════════════════
Final: ✅ PASSED (99.8% quality)
```

Done When

- [ ] Script runs without errors
- [ ] All checks pass
- [ ] You can confidently say "my data is clean"

Phase B: FastAPI Backend (8–10 hours)

What: Build a REST API so your dashboards and ML models can talk to your data stores. **Why:** Think of FastAPI as the "waiter" in a restaurant. The dashboard (customer) asks for data, the API (waiter) goes to the kitchen (InfluxDB/MinIO), gets what's needed, and serves it back neatly.

Folder Structure to Create

```
src/app/api/
├── __init__.py          ← Makes this a Python package (empty file)
├── main.py              ← The "restaurant entrance" — starts the FastAPI app
├── routes/
│   ├── __init__.py
│   ├── sensor_data.py   ← Endpoints for sensor readings (GET latest, GET history)
│   ├── analytics.py     ← Endpoints for daily summaries, aggregates
│   └── ml.py            ← Endpoints for ML predictions (built in Phase D)
├── schemas.py           ← Defines the shape of requests/responses (like a menu)
├── dependencies.py      ← Shared connections to InfluxDB and MinIO
└── utils.py             ← Helper functions (date parsing, formatting)
```

Key Endpoints to Build (in order)

Round 1 — Health & Basics (build these first):

Endpoint	Purpose	Beginner Note
`GET /api/v1/health`	Check if API + databases are up	Always build this first — it's your "is everything alive?" check
`GET /api/v1/sensors/latest`	Latest reading from all 42 motes	Returns one row per mote with the most recent values
`GET /api/v1/sensors/{mote_id}/latest`	Latest reading for one mote	Same as above but filtered to one sensor

Round 2 — Time-Series Queries:

Endpoint	Purpose	Beginner Note
`GET /api/v1/sensors/readings?mote_id=1&start=...&end=...`	Historical readings for a mote	This powers your trend charts in the dashboard
`GET /api/v1/sensors/{mote_id}/stats?start=...&end=...`	Aggregated stats (avg, min, max, stddev)	Used for summary cards and KPI tiles

Round 3 — Analytics:

Endpoint	Purpose
`GET /api/v1/analytics/daily-summary?date=2026-02-19`	One-day overview across all motes
`GET /api/v1/motes`	List all 42 motes with status

How to Build in VS Code

1. Install dependencies: `pip install fastapi uvicorn pydantic`
2. In `main.py`: Create the FastAPI app, add CORS middleware, include route files
3. In `dependencies.py`: Create InfluxDB and MinIO client instances using `.env` variables
4. In `schemas.py`: Define Pydantic models for your request/response shapes
5. Build routes one at a time, test each with the auto-generated Swagger docs at `http://localhost:8000/docs`
6. Run with: `uvicorn src.app.api.main:app --reload`

How to Test Each Endpoint

FastAPI auto-generates interactive documentation. Open `http://localhost:8000/docs` in your browser, click any endpoint, hit "Try it out", and see the response live. No Postman needed.

Done When

- [ ] `GET /health` returns `{"status": "healthy"}`
- [ ] `GET /sensors/latest` returns readings for all 42 motes
- [ ] `GET /sensors/readings` returns historical data for any mote + date range
- [ ] Swagger docs render at `/docs`

Phase C: ML Model Training (6–8 hours)

What: Train an Isolation Forest anomaly detection model on your historical sensor data. **Why:** This is the "intelligence" of your system — it learns what "normal" looks like so it can flag unusual readings.

Folder Structure to Create

```
src/app/ml/
├── __init__.py
├── config.py              ← ML settings (contamination rate, feature list, model paths)
├── models/
│   ├── __init__.py
│   └── anomaly_detector.py    ← The Isolation Forest class (train, predict, save, load)
├── training/
```

```
│   ├── __init__.py
│   ├── data_loader.py        ← Pull data from MinIO Parquet files into a pandas DataFrame
│   └── feature_engineer.py   ← Create features (hour_of_day, day_of_week, rolling averages)
└── serving/
    ├── __init__.py
    └── predictor.py          ← Load a saved model and make predictions

models/                       ← Saved model files (at project root)
├── anomaly_detector_v1_20260219.pkl
└── model_registry.json       ← Tracks which model version is "current"

scripts/
└── train_model.py            ← One-click training script
```

The Training Pipeline (Step by Step)

Think of this as a **cooking recipe with 5 steps:**

Step 1 — Gather Ingredients (Data Loading)

- Source: MinIO Parquet files (cold storage, full historical coverage)
- Read all Parquet files into one big pandas DataFrame
- Expect ~1.3M rows with columns: `timestamp, mote_id, temperature, humidity, light, voltage`

Step 2 — Prep the Ingredients (Feature Engineering)

- From the timestamp, extract: `hour_of_day` (0-23) and `day_of_week` (0-6)
- These matter because sensor patterns change by time of day (lights off at night, temperature drops)
- Optional but valuable: rolling mean and rolling standard deviation over the last 6 readings per mote
- Final feature set: `temperature, humidity, light, voltage, hour, day_of_week` (6 features)
- Handle missing values: drop rows where any sensor reading is null

Step 3 — Cook (Model Training)

- Algorithm: `IsolationForest` from scikit-learn
- Key parameters:
  - `contamination=0.05` → Tells the model "expect about 5% of readings to be anomalies"
  - `n_estimators=100` → Number of isolation trees (more = better but slower; 100 is a good default)
  - `random_state=42` → Makes results reproducible (same data → same model every time)
- Before training: scale features using `StandardScaler` (converts all values to similar ranges so no single feature dominates)
- Fit the model on the scaled historical data

Step 4 — Taste Test (Model Evaluation)

- Run predictions on the training data itself
- Check: What percentage did it flag as anomalies? (Should be close to 5%)
- Log training metrics: number of samples, features, anomaly ratio, training time
- If anomaly ratio is wildly off (like 30%), your contamination parameter needs adjusting

Step 5 — Package and Label (Save Model)

- Save the trained model + scaler using `joblib` (a Python library for saving ML objects)
- Filename format: `models/anomaly_detector_v1_YYYYMMDD.pkl`
- Update `models/model_registry.json` with the new version, timestamp, and metrics
- This registry tells the API which model file to load

What the Training Script Does

`scripts/train_model.py` orchestrates all 5 steps:

1. Load data from MinIO
2. Engineer features
3. Train Isolation Forest
4. Evaluate and print metrics
5. Save model artifact + update registry

Run it with: `python scripts/train_model.py`

Expected Training Output

```
SIEIS ML Training Pipeline
═══════════════════════════════════════

Loading data from MinIO...     1,312,456 records loaded
Engineering features...        6 features created
Handling missing values...     1,280,000 clean records
Scaling features...            StandardScaler fitted
Training Isolation Forest...   Done in 12.3 seconds
Evaluation:
  Samples:       1,280,000
  Features:      6
  Anomalies:     64,000 (5.0%)
  Model size:    8.2 MB
Saving model...                models/anomaly_detector_v1_20260219.pkl
Registry updated.
```

```
═══════════════════════════════════════
✅ Training complete!
```

Done When

- [ ] `scripts/train_model.py` runs end-to-end without errors
- [ ] A `.pkl` file exists in `models/`
- [ ] `model_registry.json` has the correct version info
- [ ] Anomaly ratio is between 3-8% (reasonable for sensor data)

Phase D: ML Inference API (4–5 hours)

What: Add prediction endpoints to your FastAPI app so dashboards can send sensor readings and get back "normal" or "anomaly". **Why:** The trained model sitting in a file is useless until something can ask it questions. The API makes the model accessible.

Analogy: Training the model is like teaching a dog to detect drugs. Deploying it is like bringing the dog to the airport and letting passengers walk past it.

What to Build

1. Predictor Service (`src/app/ml/serving/predictor.py`)

- On startup: reads `model_registry.json`, loads the latest `.pkl` file
- Exposes a `predict()` method that takes sensor values and returns anomaly score + label
- Keeps the model in memory (no file I/O per prediction = fast)

2. ML API Route (`src/app/api/routes/ml.py`)

Endpoint	Method	Purpose
`/api/v1/ml/predict/anomaly`	POST	Send one reading → get anomaly score
`/api/v1/ml/predict/batch`	POST	Send multiple readings → get scores for all
`/api/v1/ml/model/info`	GET	Returns current model version, training date, metrics
`/api/v1/ml/retrain`	POST	Triggers a retrain (Phase F)

How a Prediction Works (End to End)

```
Dashboard sends:
  POST /api/v1/ml/predict/anomaly
  Body: {"temperature": 45.2, "humidity": 30.1, "light": 250, "voltage": 2.8}
```

```
API receives → extracts features → adds hour/day_of_week → scales with saved scaler
  → runs through Isolation Forest → gets anomaly score

API returns:
  {
   "is_anomaly": true,
   "anomaly_score": -0.72,
   "severity": "high",
   "details": "Temperature 45.2°C significantly above normal range (18-28°C)"
  }
```

How Severity Works

Score Range	Label	Meaning
-0.3 to 0.0	Low	Slightly unusual but likely fine
-0.6 to -0.3	Medium	Worth investigating
Below -0.6	High	Definitely anomalous

(These thresholds are tunable in `src/app/ml/config.py`)

How to Build in VS Code

1. Create `predictor.py` — a class that loads model on init, has a `predict()` method
2. In `dependencies.py` — instantiate the predictor as a singleton (loaded once, shared across requests)
3. Create `routes/ml.py` — define the endpoints, call predictor methods
4. Register the ML routes in `main.py`
5. Test via Swagger: POST a sample reading, verify you get an anomaly response

Done When

- [ ] `POST /ml/predict/anomaly` returns a valid prediction
- [ ] `GET /ml/model/info` shows model version and training metrics
- [ ] Prediction latency is under 50ms per reading
- [ ] Batch endpoint handles 100+ readings at once

Phase E: Streamlit Dashboards (10–12 hours)

What: Build two visual dashboards — one for real-time monitoring (InfluxDB) and one for historical analysis (MinIO). **Why:** Dashboards are how humans consume data. All the infrastructure you've built becomes useful when someone can actually see trends, anomalies, and patterns on a screen.

Folder Structure to Create

```
src/app/dashboard/
├── app.py                ← Main Streamlit entry point with sidebar navigation
├── pages/
│   ├── 1_realtime.py     ← Hot path dashboard (InfluxDB data)
│   ├── 2_historical.py   ← Cold path dashboard (MinIO data)
│   ├── 3_ml_insights.py  ← ML predictions and anomaly analysis
│   └── 4_system_health.py ← Pipeline monitoring (Kafka lag, DB status)
└── utils/
    ├── __init__.py
    ├── influx_client.py  ← Helper to query InfluxDB for dashboard
    ├── minio_client.py   ← Helper to read MinIO Parquet for dashboard
    └── api_client.py     ← Helper to call your FastAPI endpoints
```

Dashboard 1: Real-Time Monitoring (pages/1_realtime.py)

Data Source: InfluxDB (last 24 hours) **Refresh:** Auto-refresh every 10-30 seconds using Streamlit's `st.rerun()`

Layout:

```
┌──────────────────────────────────────────────────────────┐
│ [KPI Row]                                                │
│ Active Motes: 42   Latest Update: 5s ago   Anomalies: 3  │
│ Avg Temp: 22.1°C   Avg Humidity: 44.5%     Rate: 40K/min │
├──────────────────────────────────────────────────────────┤
│ [Mote Grid]                                              │
│ A color-coded grid of all 42 motes                       │
│ Green = normal, Yellow = warning, Red = anomaly          │
│ Click a mote → shows its detail below                    │
├──────────────────────────────────────────────────────────┤
│ [Trend Charts]                                           │
│ Line chart: Temperature over last 24h (selected mote)    │
│ Line chart: Humidity over last 24h (selected mote)       │
├──────────────────────────────────────────────────────────┤
│ [Anomaly Timeline]                                       │
│ Scatter plot: Anomaly scores over last 24h               │
│ Red dots = flagged anomalies                             │
└──────────────────────────────────────────────────────────┘
```

Dashboard 2: Historical Analysis (pages/2_historical.py)

Data Source: MinIO Parquet files (full date range) **Refresh:** On-demand (user selects date range)

Layout:

```
┌──────────────────────────────────────────────────────────┐
│ [Filters]                                                │
│ Date Range Picker: [Jan 19] to [Feb 19]  Mote: [All ▼]   │
├──────────────────────────────────────────────────────────┤
│ [KPI Row]                                                │
│ Total Records: 1.3M    Days Covered: 31    Avg Quality: 99% │
├──────────────────────────────────────────────────────────┤
│ [Daily Trends]                                           │
│ Line chart: Daily average temperature across all motes   │
│ Line chart: Daily average humidity                       │
├──────────────────────────────────────────────────────────┤
│ [Seasonality Heatmap]                                    │
│ X-axis: hour of day (0-23)                               │
│ Y-axis: day of week (Mon-Sun)                            │
│ Color: average temperature → reveals daily/weekly patterns │
├──────────────────────────────────────────────────────────┤
│ [Distribution Histograms]                                │
│ 4 histograms: temperature, humidity, light, voltage      │
│ Shows the spread of values across the entire history     │
└──────────────────────────────────────────────────────────┘
```

Dashboard 3: ML Insights (pages/3_ml_insights.py)

Data Source: Your FastAPI ML endpoints **Purpose:** Show what the model is finding

Layout:

```
┌──────────────────────────────────────────────────────────┐
│ [Model Info Bar]                                         │
│ Model: Isolation Forest v1    Trained: Feb 19    Status: ✅ │
├──────────────────────────────────────────────────────────┤
│ [Live Anomaly Feed]                                      │
│ Table: Recent anomalies (timestamp, mote, metric, severity) │
├──────────────────────────────────────────────────────────┤
│ [Anomaly Score Distribution]                             │
│ Histogram of scores: most readings cluster near 0 (normal) │
│ Tail stretching to -1 (anomalous)                        │
├──────────────────────────────────────────────────────────┤
│ [Manual Prediction Tool]                                 │
│ 4 sliders: temperature, humidity, light, voltage         │
│ "Predict" button → calls API → shows result              │
├──────────────────────────────────────────────────────────┤
│ [Retrain Button]                                         │
│ "Retrain Model" → triggers /api/v1/ml/retrain            │
│ Shows progress and new model metrics when done           │
└──────────────────────────────────────────────────────────┘
```

How to Build in VS Code

1. Install: `pip install streamlit plotly`
2. Start with `app.py` — just the sidebar navigation and page routing
3. Build `1_realtime.py` first (most visible, most impressive for a demo)
4. Use `st.metric()` for KPI cards, `plotly.graph_objects` for charts
5. Query InfluxDB directly from the dashboard OR call your FastAPI endpoints via `requests`
6. Run with: `streamlit run src/app/dashboard/app.py`
7. Opens at `http://localhost:8501`

Done When

- [ ] All 4 pages load without errors
- [ ] Real-time page auto-refreshes and shows live data
- [ ] Historical page renders trend charts for any selected date range
- [ ] ML page shows anomaly predictions and model info
- [ ] Charts render in under 2 seconds

Phase F: Incremental Retraining (3–4 hours)

What: A script + API endpoint that retrains the model on new data without starting from scratch. **Why:** Your simulator keeps generating new data daily. The model trained on Jan 19 – Feb 19 data won't know about Feb 20+ patterns. Retraining incorporates new patterns.

Analogy: Updating a Textbook Edition You don't rewrite the whole textbook — you add a new chapter with recent discoveries and update the edition number. Similarly, incremental retraining pulls recent data, re-fits the model, saves a new version, and swaps it into the running API.

What to Build

1. Retrain Script (`scripts/retrain_model.py`)

This script does:

1. Read the current model version from `model_registry.json`
2. Pull the last 7–30 days of data from MinIO (configurable)
3. Engineer features (same pipeline as initial training)
4. Train a NEW Isolation Forest on this data
5. Save as `models/anomaly_detector_v{N+1}_YYYYMMDD.pkl`
6. Update `model_registry.json` to point to the new version
7. Print comparison: old model metrics vs new model metrics

2. API Endpoint (`POST /api/v1/ml/retrain`)

When called:

- Runs the retrain pipeline in the background
- Returns immediately with `{"status": "retraining_started", "estimated_time": "30s"}`
- On completion: the predictor service automatically reloads the new model
- Next prediction request uses the updated model

3. Model Hot-Swap in Predictor

The predictor service checks `model_registry.json` periodically (every 60 seconds) or reloads on demand after retraining. This way you don't need to restart the API server.

When to Retrain

Trigger	How
After loading new data	Run `python scripts/retrain_model.py` manually
Weekly maintenance	Run the script every Monday
From dashboard	Click "Retrain Model" button on ML Insights page
Via API	`POST /api/v1/ml/retrain`

Done When

- [ ] `scripts/retrain_model.py` creates a new model version
- [ ] `model_registry.json` updates correctly
- [ ] API's next prediction uses the new model without restart
- [ ] Dashboard "Retrain" button works end-to-end

Phase G: Integration & Polish (4–5 hours)

What: Wire everything together, add tests, and make it demo-ready.

Integration Checklist

Task	What It Means
End-to-end data flow test	Simulator → Kafka → Consumer → InfluxDB + MinIO → API → Dashboard (all connected)
ML pipeline test	MinIO data → Training → Model saved → API loads it → Dashboard shows predictions

Task	What It Means
Error handling	What happens when InfluxDB is down? When model file is missing? Add graceful fallbacks
Logging	Add `print()` or Python `logging` statements at key points so you can debug issues
Requirements update	Make sure `requirements.txt` has ALL packages: `fastapi, uvicorn, streamlit, plotly, scikit-learn, joblib, pandas, pyarrow, influxdb-client, minio, pydantic, python-dotenv`

Test Scenarios to Verify

1. **Start fresh:** `docker-compose down && docker-compose up -d` → wait 30s → run `verify_influxDb.py` → data appears
2. **Train model:** `python scripts/train_model.py` → `.pkl` file created → metrics printed
3. **Start API:** `uvicorn src.app.api.main:app --reload` → `/docs` loads → `/health` returns OK → `/ml/model/info` shows model version
4. **Test prediction:** POST to `/ml/predict/anomaly` with sample values → get anomaly score back
5. **Start dashboard:** `streamlit run src/app/dashboard/app.py` → all 4 pages render → charts show data
6. **Retrain:** Click "Retrain" on dashboard → new model version appears → predictions use new model

Final Project Structure

```
SIEIS/
├── .env                        ← Environment variables (tokens, URLs)
├── docker-compose.yml          ← Infrastructure (InfluxDB, MinIO, Kafka, Simulator, Consumer)
├── requirements.txt            ← All Python dependencies
├── Dockerfile.consumer         ← Consumer container
├── Dockerfile.simulator        ← Simulator container
├── data/                       ← Raw datasets
├── models/                     ← Saved ML models (.pkl files)
│   ├── anomaly_detector_v1_20260219.pkl
│   └── model_registry.json
├── scripts/
│   ├── load_historical_data.py    ← Phase 1 (done)
│   ├── verify_influxDb.py         ← Phase 1 (done)
│   ├── verify_minio_storage.py    ← Phase 1 (done)
│   ├── validate_data_quality.py   ← Phase A
│   ├── train_model.py             ← Phase C
│   └── retrain_model.py           ← Phase F
├── src/app/
│   ├── simulator/              ← Phase 1 (done)
│   ├── consumer/               ← Phase 1 (done)
│   ├── config/                 ← Phase 1 (done)
│   ├── api/                    ← Phase B + D
```

```
│   │   ├── main.py
│   │   ├── routes/
│   │   │   ├── sensor_data.py
│   │   │   ├── analytics.py
│   │   │   └── ml.py
│   │   ├── schemas.py
│   │   ├── dependencies.py
│   │   └── utils.py
│   ├── ml/                    ← Phase C + D + F
│   │   ├── config.py
│   │   ├── models/
│   │   │   └── anomaly_detector.py
│   │   ├── training/
│   │   │   ├── data_loader.py
│   │   │   └── feature_engineer.py
│   │   └── serving/
│   │       └── predictor.py
│   └── dashboard/             ← Phase E
│       ├── app.py
│       ├── pages/
│       │   ├── 1_realtime.py
│       │   ├── 2_historical.py
│       │   ├── 3_ml_insights.py
│       │   └── 4_system_health.py
│       └── utils/
│           ├── influx_client.py
│           ├── minio_client.py
│           └── api_client.py
├── tests/
│   ├── test_api_endpoints.py
│   ├── test_ml_pipeline.py
│   └── test_data_quality.py
└── Documentation/
    ├── ARCHITECTURE.md
    ├── ML_MODELS.md
    ├── ML_ACTION_PLAN.md
    ├── PROJECT_ROADMAP.md
    ├── DEPLOYMENT.md
    └── NEXT_STEPS.md
```

Master Timeline

Phase	Task	Hours	Depends On	Key Deliverable
A	Data Validation	2–3h	Data loaded ✅	Quality report passes
B	FastAPI Backend	8–10h	Phase A	Swagger docs + working endpoints
C	ML Model Training	6–8h	Phase A	Saved `.pkl` model file
D	ML Inference API	4–5h	Phase B + C	`/ml/predict/anomaly` works
E	Streamlit Dashboards	10–12h	Phase B + D	4 dashboard pages rendering
F	Incremental Retraining	3–4h	Phase C + D	Retrain button creates new model version
G	Integration & Polish	4–5h	All above	Full end-to-end demo works
	TOTAL	**37–47h**		

Suggested Daily Schedule (If Working Full Days)

Day	Focus	Outcome
Day 1	Phase A + start Phase B	Data validated, API skeleton running
Day 2	Finish Phase B	All sensor/analytics endpoints working
Day 3	Phase C	Trained ML model saved
Day 4	Phase D + start Phase E	Prediction API working, real-time dashboard started
Day 5	Finish Phase E	All 4 dashboard pages rendering
Day 6	Phase F + Phase G	Retraining works, full integration test passes

---

VS Code Tips for This Project

1. **Open the project folder** as workspace root (the folder with `docker-compose.yml`)
2. **Use the integrated terminal** (Ctrl+`) — run Docker, Python, and Streamlit from here
3. **Install Python extension** by Microsoft — gives you linting, autocomplete, debugging
4. **Run multiple terminals:** One for Docker logs, one for FastAPI, one for Streamlit
5. **Use `.env` file support:** Install "DotENV" extension for syntax highlighting

6. **Debug FastAPI:** Add a VS Code launch config for `uvicorn` with `--reload`

Terminal Layout While Developing

```
Terminal 1: docker-compose logs -f          ← Watch infrastructure
Terminal 2: uvicorn src.app.api.main:app --reload   ← API server
Terminal 3: streamlit run src/app/dashboard/app.py   ← Dashboard
Terminal 4: (free for running scripts)      ← Training, validation, etc.
```

Common Pitfalls to Avoid

Pitfall	Why It Happens	How to Avoid
Model training fails with NaN errors	Missing values in data weren't handled	Always `dropna()` or `fillna()` before training
API returns 500 errors	Model file not found or wrong path	Use absolute paths and check `model_registry.json` on startup
Dashboard is slow	Querying all 1.3M records on every refresh	Use aggregation queries (hourly/daily averages) instead of raw data
Retraining overwrites old model	File naming doesn't include version	Always use timestamps in filenames: `model_v2_20260220.pkl`
Features mismatch between training and prediction	Training used 6 features but API sends 4	Save the feature list alongside the model and validate on prediction
Docker containers out of memory	InfluxDB + MinIO + Kafka eat RAM	Set memory limits in `docker-compose.yml`, restart containers if needed

---

This plan covers your complete path from current state to a fully working ML-powered IoT dashboard. Build each phase sequentially — each one builds on the previous. When in doubt, verify the previous phase is solid before moving forward.